In-Flow Peer-Review of Tests in Test-First Programming

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Kathi Fisler
WPI
kfisler@cs.wpi.edu

ABSTRACT

Test-first development and peer review have been studied independently in computing courses, but their combination has not. We report on an experiment in which students in two courses conducted peer review of test suites while assignments were in progress. We find strong correlation between review ratings and staff-assessed work quality, as well as evidence that test suites improved during the review process. Student feedback suggests that reviewing had some causal impact on these improvements. We describe several lessons learned about administering and assessing peer-review within test-first development.

Categories and Subject Descriptors: K.3.2 [Computers and Education]: Computer and Information Science Education

Keywords: Peer-review, Test-first development

1. INTRODUCTION

In many disciplines with a craft component, apprentices learn from observing masters. Observing someone else's work and producing a critique of it helps develop critical facilities, which in turn can be applied to one's own efforts. However, programming neophytes usually lack the skills to even examine the work of a master (e.g., a beginning programming student likely has no hope of understanding the Linux kernel). What can students possibly review?

One very useful object of review is the work of fellow students. Reviewing peer work eliminates numerous variables that would otherwise make review difficult or even impossible: classmates typically use the same language, the same features, and work on the same problem. Indeed, studying the work of classmates on a problem can even help a student understand the problem better. If peer-review is then done *while* a problem is in progress, students can improve their understanding in a way that is immediately useful—this incentivizes reviewing effort. We call this *in-flow peer-review.*

One challenge with in-flow peer review lies in identifying sufficiently mature artifacts for early review. As educators who use *test-first* programming,[1] we find the point at which tests are submitted provides a natural review boundary. This moment is especially conducive for several reasons:

- Tests represent a student's understanding of a problem. Therefore, they can reveal incorrect and incomplete understanding. Reviewing at this stage therefore has the potential to set students on the right path for the remainder of the problem.
- Because tests—especially unit tests—are both (a) concrete and (b) usually understandable in isolation from one another, they can be easier to comprehend (and thus review well) than programs, which tend to be both abstract and full of subtle interdependencies.
- In some professional settings [12], tests are actually used as a means of communication. Submitting and reviewing tests reinforces this use.
- Reading others' tests might improve a student's abilities as a tester. Because testing is a vital part of modern software development,[2] and even a profession in itself, this enriches a career skill.

Our vehicle for in-flow peer-review of tests is CaptainTeach, an on-line environment designed to facilitate this process [13]. We have used CaptainTeach in two collegiate courses in the past year. Though CaptainTeach was used to provide feedback for more than test suites, including implementations, in this paper we focus on its effect on testing.

In-flow peer-review of tests raises several natural questions. Do students submit tests and reviews early enough to be of use? Do their test suites evolve during review? Are the reviews accurate and helpful? This paper explores the viability of in-flow peer review through these questions, while simultaneously reporting observations about the process. We believe this is the first study of in-flow peer-reviewing of any significant artifacts. In addition, while other projects have incorporated tests into peer-review (Section 5), we are not aware of prior work in which students review others' test suites themselves as standalone artifacts.

[1] We use the term test-*first* because students are asked to write tests before writing code. We do not follow the stronger test-*driven* [1] practice, which prohibits writing any code before tests have first been written and found to fail.

[2] One of the spaces at Google most likely to get an employee's attention—namely, the blank spaces that are stared at in restrooms—features a series of articles known as TotT: Testing on the Toilet (`http://googletesting.blogspot.com/2007/01/introducing-testing-on-toilet.html`). That is, when a major software company can best get its employees' attention, it emphasizes testing.

ICER'14, August 11-13, 2014, Glasgow, Scotland, UK.
Copyright 2014 ACM 978-1-4503-2755-8/14/08 ...$15.00.
http://dx.doi.org/10.1145/2632320.2632347.

2. ASSESSING TESTING

Traditionally, test suites are evaluated by code coverage. However, coverage is relative to a particular implementation, and is thus not meaningful for tests written before code. Instead, we must measure how well a test suite reflects the problem's intent. We use two criteria for this: *correctness* and *thoroughness*. Intuitively, correctness assesses whether tests yield answers that are consistent with the problem statement, while thoroughness assesses whether a set of tests covers the interesting inputs to a problem. In defining these criteria more rigorously, we will use two terms: a *test case* is a particular input-output pair that checks a single point in the space of a program's (or function's) behaviors; a *test suite* is a collection of test cases.

Students in our courses assessed correctness and thoroughness through peer-review; Section 3.1 explains the process. Our course staff independently assessed these criteria in assigning grades: in one course, this assessment was automated, as described in the rest of this section. (Section 4.3 compares the student- and staff-assessments.)

Course staff assess correctness using reference implementations, which we call *gold solutions*. An individual test case is correct if the gold solution returns the test case's output when given its input. Correctness is, however, insufficient to measure an entire test suite. To build intuition, suppose the programming task is to implement sorting. A weak test suite might check only a few small examples, and perhaps even only ones that are already sorted. These will all pass the gold solution, but do not represent a deep understanding of the nuances of the problem. Thoroughness fills this gap.

Staff assess thoroughness using buggy, or *coal*, solutions. Coal solutions have the same interface as the gold solution, and are usually designed to reflect a particular error. The deviation from correct behavior could be large, or it could be small and subtle. Coal solutions for sorting might include: the identity function (which "works" on already sorted lists); one that reverses its input; one that permutes the input list randomly; one that fails on empty lists; one that drops duplicate entries; one that adds arbitrary new elements (but in sorted order) to the output; and so on. A test case detects a coal solution either because (1) it expects a particular concrete answer but the coal solution yields a different answer or an exception, or (2) it expects an exception but the coal solution yields an answer or a different exception.

A test suite should act as a classifier, accurately labeling each program it is run against as gold or coal. This check can be automated, computing a test suite's grade as a function of the percentages of correct test cases and of coals detected. Section 3.3 describes the formulas used in the courses in this study. Of course, a student's test suite might also target aspects of her particular implementation: our gold/coal methodology works with any suite whose tests match the interfaces defined for the problem.

3. EXPERIMENTAL SETUP

We used CaptainTeach in two undergraduate courses in the fall semester of 2013 at Brown University (USA). One, which we label CS1.5, is an accelerated introduction to data structures and algorithms for first-year students that compresses the first-year curriculum into a semester, roughly like a "honors" course (though it is open to all students, and students place into it by doing extra homeworks in a regular introductory course). The other, which we dub CSPL, is an upper-level course in programming languages, attended by second-year undergraduate through graduate-level students. All programming was done in Pyret (`pyret.org`).

In CS1.5 the homeworks consisted of algorithmic tasks, often embedded in larger systems. Many assignments thus had multiple, ordered problems, each requiring both a test suite and an implementation. In CSPL, all assignments consisted of the implementation of a single significant function (interpreter, type-checker, garbage collector, etc.), and required a single test suite. Within each problem in both courses, the test suites and implementations formed distinct *steps* that had to be submitted *separately and in order* (CaptainTeach enforced this). In both courses, the staff provided the interfaces against which students wrote tests. While we *encouraged* students to write tests before implementations, we could not enforce the order in which they actually did their (offline) work. For each problem, students could submit one set of tests and one implementation for review before the deadline: we term these *initial* submissions. Subsequent submissions were recorded but not sent out for review. A student's *final* submission (for each of tests and implementation) was the last made before the deadline.

For various reasons (including the complexity of the assignments), we used gold/coal solutions in CSPL but not in CS1.5. Therefore, some of our analyses cover both courses, while others look only at CSPL. In particular, any reference to measured test-suite quality is limited to CSPL.

3.1 The Review Process

After submitting each step, students wrote reviews of work others did for that same step. With 50% probability, one review was of a staff-written gold or coal solution (with equal chance of being gold or coal). Other works presented for review were from classmates, chosen by picking the submissions with the fewest number of assigned reviews, breaking ties by picking earlier submissions. To have something to review for the first three students to submit, we seeded the system with three fake student submissions that were not designated as either gold or coal.

CSPL students wrote three reviews per step, while CS1.5 students typically wrote only two (they wrote three on one assignment). Each CS1.5 assignment had multiple interacting functions; students had to submit tests and bodies for each function before moving on to the next. Because these assignments had more steps, we asked for fewer reviews.

For test-suite reviews, students were asked to judge both correctness and thoroughness: each had a Likert-scale rating (required for submission) and a comment box for elaborating on their rating (not required for submission). The Likert scale had 6 ratings, from Strongly Disagree to Strongly Agree (with three positive and three negative options, and no neutral option). The two prompts were:

1. "These tests correctly reflect the desired behavior."
2. "These tests are representative of the possible inputs."

Submission for later steps was disabled while students had outstanding reviews to complete. Upon receiving a review, CaptainTeach sent email to the author of the test suite with a link a to page that displayed the review.

Despite other work that argues for detailed rubrics [10], we intentionally left ours unstructured. Because little prior work considers reviews of tests, we wanted baseline data on in-flow test reviewing that we could build on in future work.

3.2 Review Feedback

Students received two kinds of feedback on their reviews:

1. When students graded gold or coal solutions, CaptainTeach gave feedback on how they were doing. If their review was inappropriate (Slightly Disagree or lower for gold, Slightly Agree or higher for coal), they were informed of the discrepancy, whereas if it was appropriate (Slightly Agree or higher to gold, Slightly Disagree or lower to coal), they were informed of our agreement.

2. When reading a review, students could provide feedback to the reviewer. Submission of feedback was optional; we report on its frequency in Section 4.6. If students gave feedback, they had to fill out a Likert scale with the prompt "This review was helpful"; they could optionally also offer free-form comments.

3.3 Grading and Motivation

In addition to written feedback from course staff, CSPL students also received a single numeric grade for their test suite. This was computed using the gold and coals. As a baseline, we took the thoroughness (proportion of coal tests caught) of a suite. We then subtracted two percentage points for *each test case that failed* when run against the gold implementation. We did this instead of counting the number or proportion of tests that passed the gold implementation since this rewards "stuffing" the test suite with redundant or simplistic tests. Penalizing for incorrect tests while using the overall thoroughness as a baseline allowed us to avoid the most obvious ways of gaming the system.

To ensure students put a genuine effort into their initial submissions, we graded both the initial submission and the final one, and gave a weight of 75% to the initial and 25% to the final. The weights were chosen so that learning from others' work from feedback could make a qualitative difference, but would not be a substitute for students doing their own work. In particular, a student who submitted a blank test suite and copied the best of the ones given to review would be sure to get a failing grade. (Nobody tried this.)

Finally, we had repeated discussions with students about the goals and design of the peer-review process and of our system of weights. Students appeared able to articulate the benefits of peer-review, and in anonymous course evaluations found the process largely beneficial (though it was regarded much more useful for some assignments than others). Even though the workload had the potential to increase (because the assignments were largely unchanged from previous years), students did not report working significantly more time. Curiously, an independent (student-run) evaluation rated CSPL substantially easier (by one point on a four-point scale) than in previous years; though this cannot necessarily be attributed to peer-review, it does suggest that peer-review did not make the course substantially harder.

4. DATA ANALYSIS

Our analysis of peer-review for testing relies on a combination of data extracted from CaptainTeach and manual coding of extracted data. We extracted the contents of all review and review-feedback forms, the time of each submission (both work and reviews), and the initial access times of each review. The course staff provided information on which submissions were seeds, to filter out in our analysis.

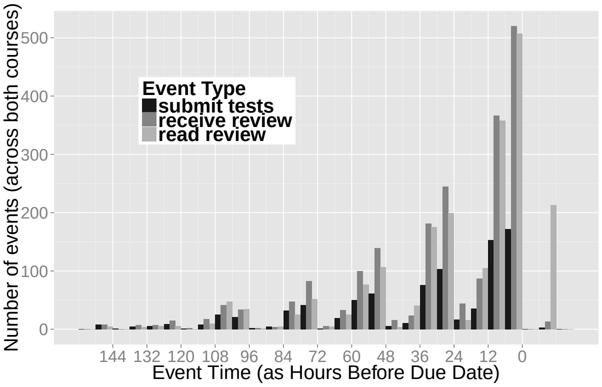


Figure 1: Timings of key events in test-suite review. Times are given as deltas from the assignment deadline, grouped in 6-hour intervals (time 0 is the deadline). Bars to the right of 0 represent late assignments. The far left bars aggregate activity earlier than 150 hours before the deadline.

For analyses that looked at comment content, we developed rubrics and manually coded data. When multiple authors coded data, we coded independently only after achieving κ of at least .75 (above .8 in most cases).

All statistical analyses were done in R version 3.0.2.[3]

Summary of Parameters.

Our analysis refers to key parameters of our data, as summarized in the following table. The first column is how many students completed the course (in parens is the number of those who started it and submitted *some* step for review). Next we list the number of assignments and number of actual problems (CS1.5 assignments had multiple problems with the same due date). The last two columns report the total number of test suites and reviews that were submitted in each course: the former excludes our good and bad seed solutions, but the latter includes reviews of those solutions (so the review count is not just a multiplier on the suites).

Summary of key parameters per course

Course	students	asgns	probs	suites	reviews
CS1.5	49 (55)	4	14	621	1565
CSPL	37 (41)	8	8	288	863

4.1 Student Behavior in the Review Process

In-flow feedback on test suites cannot be helpful unless students engage in reviewing early enough to affect implementations. Our courses did not have a separate due date for the initial tests: students were free to submit their initial tests just before the deadline (though they had to leave at least a few minutes to submit test-suite reviews before being able to submit their implementations). Our analysis therefore begins with submission times of initial test suites and reviews, as well as access times on test-suite reviews.

Figure 1 shows distributions of when students submitted their initial test suites, received reviews, and read reviews.

[3] Data and scripts for this paper are available at `http://cs.brown.edu/research/plt/dl/icer2014ct/`.

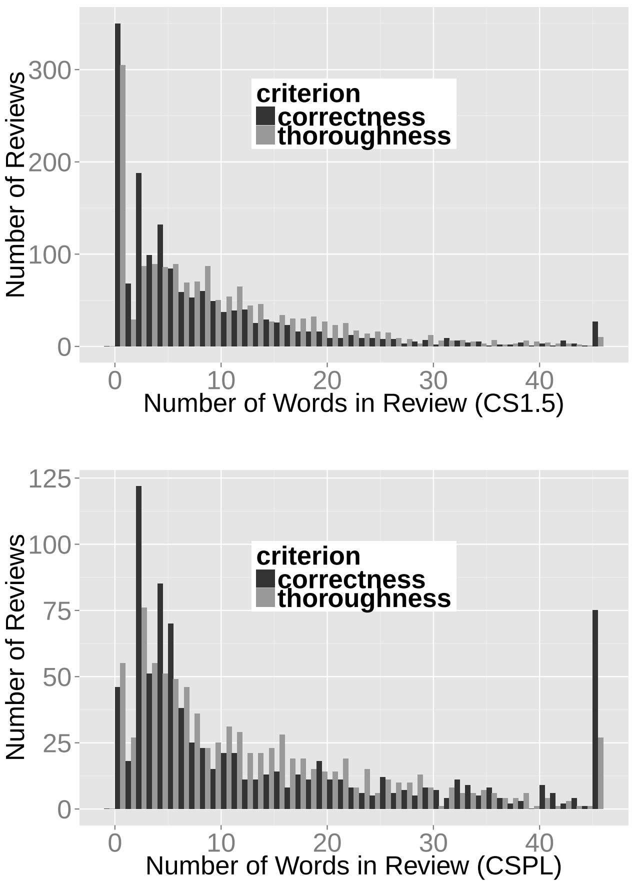


Figure 2: Word counts of review comments. The right bar aggregates comments of 50 or more words.

The data are clustered by 6-hour intervals leading to the deadline.[4] As each submission received 2–3 reviews, there are fewer submission events in the graph than receive-review or read-review events. We present a combined plot over both courses as their individual graphs were similar.

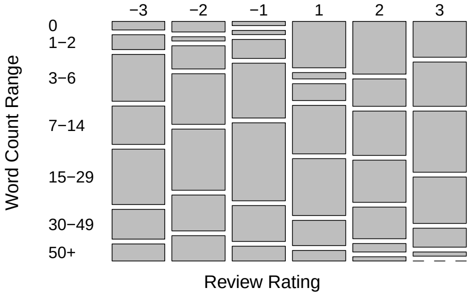
The graph shows that at least half of the test-suite submissions occurred at least 24 hours prior to the deadline. The bars for receive-review and read-review are fairly similar across the time intervals (though each class has a set of reviews that got read only after the assignment deadline had passed). There were only 6 instances in which students in CS1.5 didn't read their reviews for a particular assignment, and only 9 such instances in CSPL. The data suggest that most students are engaged to some extent in the review process, with some sufficiently engaged to submit work at least two days in advance. In particular, enough students took test-case reviews seriously that we can meaningfully explore the content and validity of their reviews.

4.2 Nature of Test-Suite Reviews

The nature of free-form review comments also reflects student engagement in reviewing. Figure 2 shows the word counts of comments given on each of the thoroughness and correctness criteria. Reviews in CSPL tend to be longer than in CS1.5, as confirmed by the following summary statistics (difference in means across the courses is significant at p=$1.6e^{-11}$ for thoroughness and p=$2.2e^{-16}$ for correctness using a Mann-Whitney-Wilcox test).

	thoroughness			correctness		
Course	mean	med	SD	mean	med	SD
CS1.5	9.48	7	9.98	7.96	4	12.29
CSPL	12.85	9	12.49	16.01	7	21.78

Students often reported (anecdotally) not knowing what to say about high-quality work. We therefore expected longer comments on lower-rated work. The following plot of review rating versus word count shows the lowest Likert rating (-3) got shorter reviews than other negative or weak ratings (-2 to 1). This suggests that students reserved their writing for test suites that had some problems but did not appear hopeless. More detailed manual analysis of the review contents would be needed to confirm this interpretation.



Of course, comment length matters less than content: short reviews can be useful, and reviews of the same length can differ widely in utility (e.g., "all looks good", "test nested scopes", and "missing many cases"). Therefore, one author manually coded all test-suite reviews for one assignment from each course (choosing assignments with subtleties that we hoped reviewing would uncover), looking *only* at the review text (hiding its rating). Inspired by Nelson and Schunn's rubrics for writing assignments in undergraduate humanities courses [11], we created a coarse-grained rubric with four variables: *abstract positive* (e.g., "this looks good"), *abstract negative* (e.g., "this looks bad"), *concrete but general* (e.g., "check more error cases"), and *concrete targeted* (e.g., "your third test returns the wrong answer"). A review could be marked with multiple variables. The counts of comments were as follows:

Course	# comm	APos	ANeg	CoGen	Targeted
CS1.5-C	96	64	3	4	20
CSPL-C	108	67	2	3	41
CS1.5-T	96	36	24	7	46
CSPL-T	108	57	6	27	34

Abstract positive remarks across both criteria (thoroughness and correctness) appeared almost exclusively in reviews with high Likert ratings; we believe this mirrors the anecdotal remarks about students not knowing what to say when reviewing good work. Concrete targeted comments occur fairly uniformly across the different Likert ratings. Within

[4] All assignments were due at midnight. The periodic dips thus correspond to the midnight–6am period, suggesting that our students might be getting some sleep!

CS1.5, abstract negative comments on thoroughness skew towards reviews with lower Likert ratings. The concrete general comments on thoroughness in CSPL have no significant relationship with review ratings.

Ideally, low-quality work would receive concrete comments that highlight its weaknesses. In CS1.5, 28 of 39 reviews that gave thoroughness ratings in the lower-half of the Likert scale included concrete targeted comments; in CSPL, 5 of 15 low-rated thoroughness reviews gave concrete targeted comments, although 9 of 15 had concrete general comments. On correctness, only 2 of 29 reviews with low ratings had no concrete targeted comments. That correctness comments are more targeted than thoroughness ones makes sense: violations of correctness would arise from individual test cases, while violations of thoroughness often arise from overlooking broad classes of program inputs.

4.3 Accuracy Of Test-Suite Reviews

To check whether students are effective reviewers, we compare their ratings to the gold/coal grades. Figure 3 plots the Likert review ratings against the gold (correctness) and coal (thoroughness) grades in CSPL. The plots show that initial submissions are generally strong, and that positive reviews outweigh negative ones. Thoroughness reviews are less concentrated in the upper-right quadrant than correctness ones. The horizontal bands in the thoroughness plot reflect the nature of the grading: thoroughness grades are a percentage of a single-digit number of coal solutions, so only a few grades are possible per assignment.

The interesting sections of these graphs are the lower-right and upper-left portions, which reflect reviews that were inconsistent with our grading assessment. Possible explanations in these cases include poor performance of the reviewer on that assignment, last-minute reviewing, weaknesses in our grading mechanisms (that failed to accurately assess work quality), or lack of engagement by the reviewer. For each of thoroughness and correctness, we manually inspected all data points with the top two Likert ratings and grades below 40, as well as points with the bottom two Likert ratings and grades above 60.

For the 43 cases of low thoroughness reviews with high grades, all but a couple of the free-form comments point to concrete situations or constructs that the test suite did not adequately exercise. This strongly suggests that this segment of the plot gets populated due to limitations in our suite of coal solutions.

For the 45 cases of high thoroughness reviews with low grades, just over half gave generic comments of the form "this is great", while a dozen stated specific criticisms and potential holes in the test suites that would be consistent with lower ratings. Those dozen might reflect review inflation. For the others, we looked at reviewer performance and review submission time, but found no patterns.

The comments for the 46 reviews with low correctness ratings but high grades are also detailed and concrete. These high grades could also be indicative of weaknesses in our tests. There are only 7 high-rated correctness reviews with low grades; the comments vary widely in style, and the sample is too small to draw inferences.

4.4 Test Suite Evolution

We are interested in how test suites evolve from the initial to the final submission, and the extent to which activities during the review process correlate with changes to the test suites. For CSPL, we can explore this by looking at the gold/coal grades on each of the initial and final submissions.

Our first question is whether test suites change, both in the number of tests and in the grade earned. Across all assignments in CSPL, the number of tests change as follows:

fewer	same	1-5	6-15	16+
28	79	89	27	16

Individual test cases can change within the same number of tests. In 33 instances, grades changed within the same number of tests (each had a change in the correctness grade; 8 also had a change in thoroughness). When tests were *removed*, it was usually because the staff clarified assumptions about valid input (e.g., interpreter inputs were guaranteed to parse) after the assignment was released.

Digging deeper, we want to know how final test suites compare to initial ones relative to each of our correctness and thoroughness criteria. Understanding which errors remained and which were fixed will help us assess the effectiveness of reviews on test suites. We omit test-suites that failed to run (due to infinite loops, timeouts, etc.) in this analysis.

Changes in Thoroughness.

We found 155 instances of coal solutions being caught in a final test suite but not the corresponding initial one. These instances occur across 75 distinct test suites from 32 unique students (across all assignments). Only 5 of these 75 cases are from situations in which the initial test suite did not run. Thus 70 final test suites improved in thoroughness from initial to final submission.

We found only 5 instances of coal solutions being caught in an initial test suite but not in the corresponding final one (this does not count 14 students whose final suites did not run, but whose initial suites had detected some coal).

Changes in Correctness.

Across the course, 158 initial test suites had at least one incorrect test case; a total of 737 individual test cases failed across these 158 suites.[5] The corresponding final test suites ran for 152 of these (that is, there were 6 total cases across the course where a test suite's final submission went into an infinite loop or otherwise failed to run on the gold solution).

We are interested in incorrect test cases from the initial test suites that remain (by string equality) in the final test suites: these represent errors that "survived" any impact of seeing reviews and other students' work before final submission. Across the initial 152 test suites with corresponding final suites, there were 690 total failing test cases. Only 65 of these test cases survived, occurring across 31 final test suites. Twenty-four of the 31 test suites still had 1-2 of their original errors; the other seven retained 3-5 errors.

The 31 test suites with unfixed failures involve 6 of the 7 automatically graded assignments in CSPL, and a total of 17 students failed to remove an incorrect test case before final submission at some point in the course. The surviving 65 incorrect test cases are a majority of the 100 individual test cases that failed across all final test suites that ran. More than half of the 31 test suites were submitted at least a day before the deadline.

[5] Because the students were not given the output of our tests, they would not necessarily know of these failures.

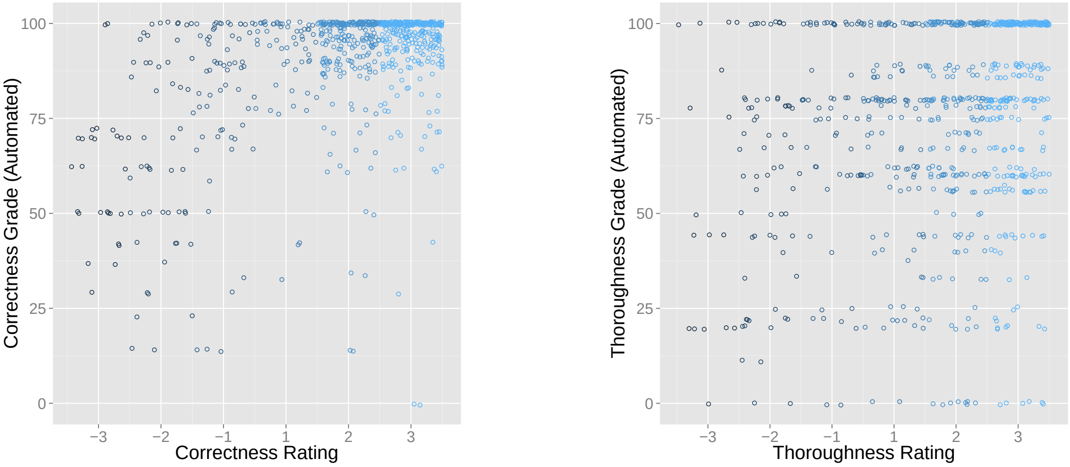


Figure 3: Review ratings versus grades on each of the correctness and thoroughness criteria (for CSPL).

4.5 Evolution Through Reviewing

The preceding data do not suggest the extent to which the review process either could have or did contribute to test-suite changes. Reviewing could have two impacts: students might receive reviews that pointed out problems (in their role as reviewees), or they might edit their tests after seeing others' work (in their role as reviewers). Our data allows us to explore both perspectives.

The following table summarizes how often CSPL test suites gained or lost detection of individual coals, and how often the review process had the *potential* to expose students to work that caught the corresponding coal. The dataset for this table contains one data point for the product of: students, assignments, and coals for that assignment.

	both	gained	lost	neither	total
# instances	1252	155	5	284	1754
as reviewer	1209	138	4	183	1590
as reviewee	1164	129	5	166	1516
neither	11	5	0	47	69

We explain the table by discussing the "gained" column. There were 155 instances of students submitting an initial test suite that failed to catch a specific coal, but a final test suite that caught the same coal. In 138 of the 155 instances, the student reviewed a test suite with a case that caught the coal. In 129 (of 155) instances, the student was reviewed by someone whose initial test suite had caught the coal. In 5 (of 155) instances, neither reviewing nor being reviewed could have exposed the student to a test that caught the coal.

We now focus on the 439 instances in which a coal was not detected initially (the "gained" and "neither" columns). We find that 89% of those who gained the coal reviewed work that caught the coal compared to 64% who did not gain the coal. This difference is significant (p=.000065, χ^2=16.02), and suggests that reviewing impacts test-suite quality. It is harder to determine impacts of reviewers having caught coals without looking at the actual review contents (which we did not have the resources to do for this paper). This may be worth doing, as 84% who gained a coal were reviewed by someone who had caught the coal, compared to 61% of those who did not gain the coal (p=.00049, χ^2=12.14).

These data are only suggestive of possible links, and by no means imply causation. Some causal information could have been obtained by asking the students directly, e.g., whether a reviewed test suite taught them anything new. However, in-flow reviewing already has the potential to be disruptive to workflow (though this proved to not be a major problem in practice; a paper about the tool [13] discusses this), and every extra prompt comes with a cost by increasing time and adding disruption. In addition, questions asked at the moment will not capture learning that occurs upon reflection. Nevertheless, teasing apart causes and effects in a setting where we don't wish to burden users too much remains a daunting task. We do, however, have one additional source of human input via the feedback comments.

4.6 Student Ratings Of Review Utility

Though providing feedback on reviews was optional, students did so more often than we expected. Out of 2428 reviews, 700 received Likert-scale feedback; 409 of the 700 also had comments (192 in CS1.5; 217 in CSPL).

Though on some social media there is a tendency to complain more than praise, Likert-ratings on reviews (when provided) were overwhelmingly positive. (This may be because students knew they were interacting with (anonymous) colleagues and friends, not unknown people on the Internet.) Across both courses, roughly 80% of feedback ratings were in the top two Likert categories; only 8% in CS1.5 and 3% in CSPL were in the lowest two Likert categories.

We manually coded all comments for indications that reviews had identified problems or led to edits in test suites (these were two of 15 categories identified during open coding of the comments). In the following table, "Ack Error" reflects feedback that acknowledged an error that was pointed out in a review, and "Will Act" captures claims that the reviewee would edit their test suite as a result of the review.

	Ack Error	Will Act	Both
# comments CS1.5	40	27	2
# authors CS1.5	20	13	2
# comments CSPL	55	30	13
# authors CSPL	25	15	10

For each course, the table reports on (a) the number of cases of each kind of comment across all review-feedback forms, and (b) the number of unique students who made such a comment. While a couple of students in each course made multiple such comments, most are more isolated cases.

The percentages of students who, unprompted, acknowledge errors and declare intent to act on reviews suggests that reviewing is successful at identifying problems in test suites. Of course, our data cannot tell us whether students would have found the errors discussed in these comments without the review process. An interesting question for future work is whether peer-review changes the set of errors caught and the time at which those errors are detected.

Reviewers can make mistakes, misinterpreting either the assignment or the work being reviewed. Feedback comments sometimes cited concrete errors on the part of reviewers. Comments that disputed specific parts of a review argued about the expected result of a test, whether a particular situation needed to be tested, or the reviewer's interpretation of the assignment. The following table summarizes reported instances of reviewer error, alongside data about whether the comments referenced the assignment:

	Dispute	Discuss Asgn	Both
# comments CS1.5	31	12	6
# reviewers CS1.5	24	10	5
# comments CSPL	41	26	19
# reviewers CSPL	24	19	14

The large number of unique reviewers (relative to the number of disputes) suggests that reviewing errors were not due to a couple of students consistently doing a poor job. This is a positive result. Some assignments in each course had interesting subtleties (without which there would be less potential value from reviewing test cases). We expected there would be some differences in interpretation. These data suggest that some discussion of those cases occurred within the reviewing process, rather than just in office hours.

5. RELATED WORK

Our paper focuses on (a) in-flow (b) peer-review of (c) tests. Little prior work covers all three aspects. We focus here on projects that combine two of them.

Kulkarni, et al. study the effects of seeing examples early on in the creative process [9]. They find that early and frequent exposure to examples (in their case, artwork) leads to participants producing drawings with more unique features (a measure of creativity). One of our hopes is that seeing examples of tests early leads students to create more diverse and effective test suites for their own programs.

Buffardi and Edwards study students' testing behaviors under test-driven-development [2]. Students could submit tests and code for automated assessment multiple times before the deadline, receiving in-flow feedback. The point at which a student's tests achieve significant coverage of her code is a key parameter in their analysis. This parameter is not meaningful in our work, since students submit (and ostensibly write) tests prior to implementations. Our two groups share an interest in leveraging early testing to improve code quality; identifying appropriate roles for peer-review in that process is an interesting open question.

McCarthy [pers. comm.], after the deadline, has students answer a questionnaire with specific test-related questions. Affirmative responses must be accompanied by references to concrete code. Other students are asked to assess these responses. All these responses are taken into account in grading. The evaluation is not in-flow. Though our gold and coal solutions simulate some of the effect of his rubrics, we may benefit from using such rubrics more directly.

Expertiza is a peer-review system with reviews of reviews, but as part of a grade rather than as helpful feedback [14]. They do not provide quantitative data on reviews of reviews. The applications of Expertiza were often on collaborative projects, where each student submitted pieces of a larger system, so students' own experience with their problem was less directly applicable to the review.

Aropä is a peer-review system that has been used in both computer science and humanities [6]. It does not support our notion of in-flow review, but it does support a review-dispute-revise loop. Since peer reviews are used as part of the grading process, accurately disputing a flawed review can lead to a better grade, so students have incentive to read their feedback with a critical eye.

Hundhausen, et al. study several variations of peer code review in their courses [7]. They do not consider testing explicitly, and do not have in-flow components. They do discuss and reflect on reviews in a group setting, as an opportunity to identify what makes for a helpful review, or what problems were seen across the class. Their work demonstrates the importance of the social aspect of peer review.

Smith, et al. have students write tests for one another as part of reviewing [16]. The tests (and any bugs found) are reported to the original authors, who evaluate the feedback, including the tests and overall testing strategy. Smith et al. do not discuss specific qualities that were evaluated of the tests and reviews, and the testing was more whole-system than in our work. When evaluating reviews, the students could also submit fixes to their program that the review identified; the authors do not provide data on how often or to what degree students exploited this.

Clark studies peer testing in larger software engineering projects across several years [3]. The testing is less automated because it involves using interactive interfaces. Afterwards, student programmers are asked to evaluate their testers with a rubric that measures helpfulness.

Søndergaard uses in-flow peer review in a course on compilers [17], though the flow is not test-first. Rather, several components of a compiler are completed in order, with review steps in between. 68% of students agreed that peer review helped them improve their work, 63% agreed it improved their ability to reflect on learning, and 89% agreed that it was useful to see other groups' solutions.

Reily, et al. have students submit test cases as part of a peer review process [15], along with Likert and open-response questions. They do not focus on reviewing test suites themselves, however, instead using tests as a kind of concrete feedback on implementations, as part of a rubric.

Gaspar, et al. surveyed students on their perceptions of Peer Testing, in which students share their test suites with classmates [5]. The time at which their students submitted tests relative to the due date is not clear, though the survey asks students about the impact of Peer Testing on their programs. Students generally perceived benefits to trading test suites. Students did not provide feedback on each others' tests, so this use of Peer Testing captures only one of the reviewing roles required of CaptainTeach students.

6. DISCUSSION AND FUTURE WORK

We have presented an analysis of what we believe is the first use of in-flow peer-review of tests. Both the in-flow process and the review of test suites are variables in this work, and we did not attempt to tease them apart. We also did not investigate the review of tests relative to the review of implementations, although we have analyzable data on both. We focus on review of tests here because they are a natural boundary for in-flow reviewing.

Our finding that students participate early and thoughtfully with little structure to reviews is promising. Our data on student engagement and feedback styles is thus a good candidate for comparison to students' behavior in more structured and enforced contexts. In future iterations, we wish to borrow rubrics from other authors (e.g., Kulkarni, et al. [10]). It would also be useful to directly ask students whether a review inspired them to make any changes.

Our analysis of comments on review helpfulness identifies a subtlety in rubrics for assessing reviews: prior rubrics [11] value concrete targeted comments over concrete but abstract ones on the theory that targeted comments are more actionable. While this theory holds for correctness, which focuses on individual tests, it does not apply as well to thoroughness, which is fundamentally about the overall structure of a test suite and is about material that might be *missing*.

While our data show test-suite quality improving during in-flow review, there are many possible causal explanations (besides chance). First, the process prompts students to submit material earlier, giving them more time to think about the problem and to revise work. Second, forcing the submission of tests for review emphasizes test quality. Third, students may benefit from reading the reviews that others wrote of their tests. Finally, students may benefit simply from having to read others' tests and articulate thoughts about them. Our system is not set up to discriminate across these different factors (and hence determine the impact of in-flow review). A future version of CaptainTeach that enables A/B testing may make this easier; e.g.: What if students are asked to write reviews but are not shown reviews of their own work? What if they submit tests to run against each others' code [4] but don't actually read the tests manually? These are worth exploring.

One feature we strongly considered incorporating was to give each reviewer the output of automated grading of submissions, to prompt them for things to comment on. We decided not to because we feared that if an automated system failed to find problems, human reviewers would be inclined to pass lightly over the test suite. The top-left and bottom-right of Figure 3 are especially interesting in this light: presenting this data might have helped the bottom-right group do a better job, but might it have made the top-left group complacent? Overall, our analysis of seeming inaccuracies between high-grade but low-rating reviews highlights weaknesses in relying on automated grading.

Finally, we want to consider the larger narrative around in-flow review of tests. Every test case has a lifecycle: it is created, possibly cloned and modified, fixed to reflect better understanding of a problem, modified to track a changing implementation, and sometimes even deleted. This lifecycle resembles Ko's model of software errors [8], and similarly can be affected by learning environments. CaptainTeach should help study the role of peer-review in this story.

Acknowledgements: Daniel Patterson helped build CaptainTeach and influenced its use. Our course staff and students engaged in this experiment with patience, humor, and insight. NSF grants and Google supported this work.

7. REFERENCES

[1] K. Beck. *Test-driven development by example*. Pearson Education, 2003.

[2] K. Buffardi and S. H. Edwards. Effective and ineffective software testing behaviors by novice programmers. In *International Computing Education Research Conference*, 2013.

[3] N. Clark. Peer testing in software engineering projects. In *Australasian Computing Education Conference*, 2004.

[4] S. H. Edwards, Z. Shams, M. Cogswell, and R. C. Senkbeil. Running students' software tests against each others' code: new life for an old "gimmick". In *SIGCSE Technical Symposium on Computer Science Education*, 2012.

[5] A. Gaspar, S. Langevin, N. Boyer, and R. Tindell. A preliminary review of undergraduate programming students' perspectives on writing tests, working with others, & using peer testing. In *ACM Conference on Information Technology Education*, 2013.

[6] J. Hamer, C. Kell, and F. Spence. Peer assessment using Aropä. In *Australasian Computing Education Conference*, 2007.

[7] C. D. Hundhausen, A. Agrawal, and P. Agarwal. Talking about code: Integrating pedagogical code reviews into early computing courses. *ACM Transactions on Computing Education*, 13(3), Aug. 2013.

[8] A. J. Ko and B. A. Myers. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing*, 16(1–2):41–84, 2005.

[9] C. Kulkarni, S. P. Dow, and S. R. Klemmer. Early and repeated exposure to examples improves creative work. In *Cognitive Science*, 2012.

[10] C. Kulkarni, K. P. Wei, H. Le, D. Chia, K. Papadopoulos, J. Cheng, D. Koller, and S. R. Klemmer. Peer and self assessment in massive online classes. *ACM Transactions on Computer-Human Interaction*, 2013.

[11] M. M. Nelson and C. D. Schunn. The nature of feedback: How different types of peer feedback affect writing performance. *Instructional Science*, 27(4):375–401, 2009.

[12] B. Pettichord and B. Marick. Agile acceptance testing. *Extreme Programming and Agile Methods*, 2418, 2002.

[13] J. G. Politz, D. Patterson, S. Krishnamurthi, and K. Fisler. CaptainTeach: Multi-stage, in-flow peer review for programming assignments. In *ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, 2014.

[14] L. Ramachandran and E. F. Gehringer. Reusable learning objects through peer review: The Expertiza approach. In *Innovate: Journal of Online Education*, 2007.

[15] K. Reily, P. L. Finnerty, and L. Terveen. Two peers are better than one: Aggregating peer reviews for computing assignments is surprisingly accurate. In *ACM International Conference on Supporting Group Work*, 2009.

[16] J. Smith, J. Tessler, E. Kramer, and C. Lin. Using peer review to teach software testing. In *International Computing Education Research Conference*, 2012.

[17] H. Søndergaard. Learning from and with peers: The different roles of student peer reviewing. In *ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 31–35, 2009.